Chapter 4

Selection Sort and Merge Sort

Copyright Oliver Serang, 2018
University of Montana Department of Computer Science

4.1 Selection sort

Selection sort is one of the simplest sorting algorithms. Its premise is simple: First, find the minimum element in the list (by visiting all indices 0, 1, 2, 3, ...). This will be result[0]. Second, find the second smallest value in the list (this is equivalent to finding the smallest value from indices 1, 2, 3, ... (because index 0 now contains the smallest value). Move this to result[1]. Third, find the smallest value from indices 2, 3, .... Move this to result[2].

Continuing in this manner, we can see that this algorithm will sort the list: the definition of sorted order is that the smallest value will be found in result[0], the second smallest value will be found in result[1], etc. This is shown in Listing 4.1.

4.1.1 Derivation of runtime #1

The runtime of this algorithm can be found by summing the cost of each step: the cost to find the minimum element in all n values, the cost to find the minimum element in n – 1 values, the cost to find the minimum element in n – 2 values, etc. This will cost n + n – 1 + n – 2 + ⋯ + 3 + 2 + 1
steps, each of which cost $\Theta(1)$ (because they are if statements, primitive
copy operations, etc.). If we pair the terms from both ends, we see that this
equals
\[
\begin{align*}
&n + 1 + n - 1 + 2 + n - 2 + 3 + n - 3 + 4 + \cdots \\
&= \frac{n+1 + n+1 + \cdots + n+1}{\frac{n}{2}} \\
&= \frac{(n+1) \cdot n}{2}.
\end{align*}
\]
Thus we see that selection sort consists of $\Theta(n^2)$ operations, each of which
cost $\Theta(1)$, and so selection sort is $\in \Theta(n^2)$.

4.1.2 Derivation of runtime #2

We can also see that selection sort consists of two nested loops, one looping
$i$ in $0, 1, \ldots n - 1$ and the other looping $j$ in $i + 1, i + 2, \ldots, n - 1$. Together,
the number of $(i, j)$ pairs visted will be $|\{(i, j) : 0 \leq i < j < n\}|$. Because we
know that $i < j$, for any set $\{i, j\}$ where $i \neq j$, we can figure out which index
is $i$ and which index is $j$ (note that sets are unordered, so $\{i, j\} = \{j, i\}$).
For example, if $\{i, j\} = \{3, 2\}$, then $i = 2$ and $j = 3$ is the only solution that
would preserve $i < j$; Therefore,
\[
|\{(i, j) : 0 \leq i < j < n\}| \\
= |\{\{i, j\} : i \neq j \land i, j \in \{0, 1, 2, \ldots, n - 1\}\}| \\
= \binom{n}{2}.
\]

\[
\binom{n}{2} = \frac{n \cdot (n-1)}{2} \in \Theta(n^2).
\]
Thus, these nested for loops will combine to perform $\Theta(n^2)$ iterations, each of which cost $\Theta(1)$ (because they only use if statements, primitive copy operations, etc.). This validates our result above that selection sort $\in \Theta(n^2)$.

Listing 4.1: Selection sort.

```
def selection_sort(arr):
    n=len(arr)
    cost=0
    for i in range(n):
        cost+=i
```

# costs $O(1)$ per $\{i, j\}$ pair where $i$ and $j$ are in $\{0, 1, \ldots n-1\}$ and
# $j>i$. this will cost $n$ choose 2, which is $\in \Theta(n^2)$.
4.2. MERGE SORT

```python
# make a local copy to modify and sort:
result = list(arr)

# compute result[i]
for i in xrange(n):
    # find the minimum element in all remaining
    min_index=i
    for j in xrange(i+1,n):
        if result[j] < result[min_index]:
            min_index=j

    # swap(result[min_index], result[j])
    temp=result[min_index]
    result[min_index]=result[i]
    result[i]=temp

    # result[i] now contains the minimum value in the remaining
    # array
print result
```

```python
print selection_sort([10,1,9,5,7,8,2,4])
```

4.2 Merge sort

Merge sort is a classic divide-and-conquer algorithm. It works by recursively sorting each half of the list and then merging together the sorted halves.

In each recursion, merge sort of size $n$ calls two merge sorts of size $\frac{n}{2}$ and then performs merging in $\Theta(n)$. Thus we have the recurrence $r(n) = 2r(\frac{n}{2}) + \Theta(n)$. Later, we will see how to solve this recurrence using the Master Theorem, but for now, we can solve it using calculus.

The overhead[^1] of each recursive call will be in $\Theta(1)$. Therefore, let us only consider the cost of merging (which eclipses the overhead of invoking the 2 recursive calls). If we draw a recursive call tree, we observe a cost of $\Theta(n)$ at the root node, and a split into two recursive call nodes, each of which will cost $\Theta(\frac{n}{2})$. These will split in a similar fashion.

[^1]: E.g., of copying parameters to the stack and copying results off of the stack.
From this we can see that the cost of each layer in the tree will be in $\Theta(n)$. For example, the recursive calls after the root will cost $\Theta(\frac{n}{2}) + \Theta(\frac{n}{2}) = \Theta(n)$. From this we see that the total runtime will be bounded by summing over the cost of each layer $\ell$:

$$\sum_{\ell=0}^{L-1} \Theta(n) = \Theta(nL) = \Theta(n \log(n)),$$

because $L$, the number of layers in the tree, will be $\log_2(n)$. The runtime of merge sort is $\in \Theta(n \log(n))$.

Listing 4.2: Merge sort.

```python
# costs r(n) = 2r(n/2) + \Theta(n) \in \Theta(n \log(n))
def merge_sort(arr):
    n=len(arr)
    # any list of length 1 is already sorted:
    if n <= 1:
        return arr

    # make copies of the first and second half of the list:
    first_half = list(arr[:n/2])
    second_half = list(arr[n/2:]

    first_half = merge_sort(first_half)
    second_half = merge_sort(second_half)

    # merge
    result = [None]*n
    i_first=0
    i_second=0
    i_result=0
    while i_first < len(first_half) and i_second < len(second_half):
        if first_half[i_first] < second_half[i_second]:
            result[i_result] = first_half[i_first]
            i_first += 1
            i_result += 1
        elif first_half[i_first] > second_half[i_second]:
            result[i_result] = second_half[i_second]
            i_second += 1
            i_result += 1
        else:
            # both values are equal:
```

```
4.2. MERGE SORT

```python
result[i_result] = first_half[i_first]
result[i_result+1] = second_half[i_second]
i_first += 1
i_second += 1
i_result += 1

# insert any remaining values:
while i_first < len(first_half):
    result[i_result] = first_half[i_first]
i_result += 1
i_first += 1

while i_second < len(second_half):
    result[i_result] = second_half[i_second]
i_result += 1
i_second += 1

return result

print merge_sort([10,1,9,5,7,8,2,4])
```